统计学习算法导论大作业报告

课程名称：统计学习算法导论	年级：2020 级	成绩：
学号：10205102417	姓名：张宏伟	

一、问题描述
二、 算法设计
2.1算法设计流程：
2.2算法和技术
2.2.1 深度学习
2.2.2 迁移学习(Transfer learning)
2.2.3 深度学习框架 Keras 和 GPU 计算
2.3模型介绍
2.3.1VGG
2.3.2Inception/GoogleNet
2.3.3ResNet模型
三、实验实施
3.1实验方法
3.1.1数据预处理
3.1.2特征提取
3.1.3参数微调
3.1.4训练
3.1.5 模型结合
3.2 核心代码介绍
3.2.1Pytorch预训练的模型：
数据增强：
特征提取
定义常量和设置随机种子：
数据读取
配置模型
定义训练过程
定义测试过程
定义预测过程
参数搜索优化
3.2.2自己定义实现CNN模型进行预测：
3.3环境介绍
3.4代码运行
四、实验结果与分析
4.1模型准确率
4.2模型结合
4.3预测结果优化
4.4合理性分析
五、结论
5.1项目总结
5.2个人总结

六、参考论文

一、问题描述

猫狗分类任务：

编写一个算法程序来对图像中是否包含狗或猫进行分类。采用本次数据集，完成猫狗分类任务

数据集：

- 训练集（类别已在文件名标注）包含10000张dogs以及10000张cats
- 测试集（无标注）包含12500张dogs和cats图片
- 验证集（文件名已标注类别）包含2500张dogs和2500张cats图片

输入输出

要求根据最终的模型，输出测试集中每张ID图片对应是狗的概率(0-1之间)。即0代表猫，1代表狗。

损失函数：

$$\mathcal{R}(\boldsymbol{W}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C}\boldsymbol{y}_c^{(n)}\log\hat{\boldsymbol{y}}_c^{(n)}$$

其中，N为测试集图片数目，c表示类别数目，y_hat为提交文件中的预测概率，y为 ground_truth(真实标签)。

二、 算法设计

2.1算法设计流程：

1. 数据预处理、增强：
    - 下载标注好的猫和狗的图像数据集。
    - 对图像进行处理，包括调整大小、标准化、裁剪等，以便于输入分类算法。
    - 用pytorch自带的Transforms对图片进行旋转,剪切,错切,缩放,翻转,边缘填充。
2. 特征提取：
    - 利用预训练的卷积神经网络（例如，VGG、ResNet、Inception等）作为特征提取器。这些网络在大规模图像数据集上进行了训练，并且能够提取出有用的图像特征。
    - 在训练过程中，冻结网络的所有权重，只保留卷积层之前的部分，并将图像输入到网络中得到特征向量。
3. 分类器训练：
    - 在特征提取器之后添加一个全连接层作为分类器。
    - 将提取到的特征向量输入到分类器中进行训练。可以使用常见的分类算法，如支持向量机（SVM）或多层感知机（MLP）来训练分类器。
    - 使用标注好的猫狗图像数据集进行训练，调整分类器的权重，使其能够准确地分类猫和狗的图像。

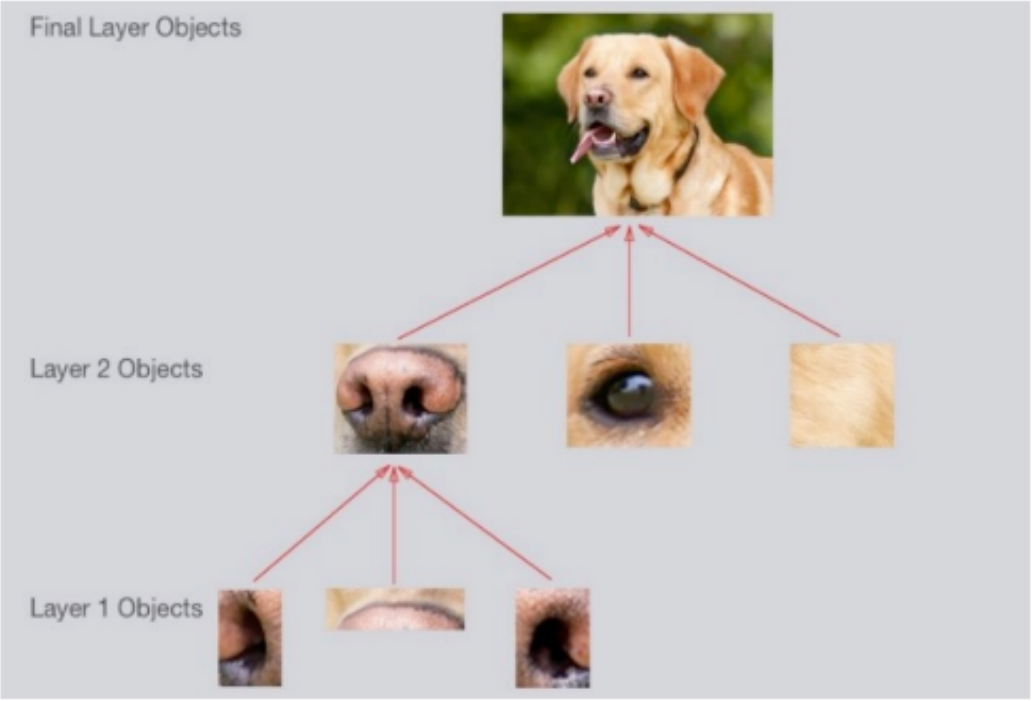
4. 模型评估与调优：
    - 使用验证集对模型进行评估，计算分类准确率、精确率、召回率等指标。
    - 根据评估结果，进行模型调优，可以尝试调整分类器的参数、增加训练数据、进行数据增强等方法来提高模型性能。
5. 预测：
    - 对测试集猫狗图像进行分类预测，记录结果。

2.2算法和技术

2.2.1 深度学习

鉴于深度卷积网络 (CNN) 在图像分类领域中的优势，将构建这样一种分类器，并逐步改善它的性能。CNN 学习识别基本的直线，曲线，然后是形状，点块，最后是图片中更复杂的物体，最终 CNN 分类器把这些大的、复杂的物体综合起来识别图片。CNN 通过正向和反向传播，自己学习识别物体，而不需要我们设定特定的特征。CNN 可能有几层网络，每个层可能捕获对象抽象层次中的不同级别。

如下图所示：**对狗图片每一层， CNN 可能识别物体的示意图**



为什么 CNN 要强于传统的计算机视觉分类方法呢? 我们知道，图像分类的传统流程包括两部分，即特征提取和分类。特征提取指的是从原始图像中提取出更具体和高级的特征，这些特征携带着具体信息并能够用于区分各个类。这种特征提取的方式是无监督模式，因为从图像的像素点中提取信息时并没有使用类别标签，例如方向梯度直方图 (HOG) 等特征提取算法。

完成特征提取后，再使用这些特征与类别标签训练一个分类模型，常用的模型有随机森林、SVM 和 LR 等。传统的图像分类流程存在一个很明显的缺点，那就是特征提取的类型必须在模型训练前确定，而不能根据图像和分类标签进行调整。如果这些提前选择的特征不足以区分各个类别，即缺乏代表性时，训练获得的模型的准确性将会是不理想的。传统图像分类流程一个较好的方法是使用多种特征提取器，然后组合这些特征以得到一种更好的特征，但这本身就是比较困难和复杂的。

而 CNN 就完全不同了，它并没有建立使用硬编码的特征提取器，而是将特征提取和分类两个模块融合在一起。CNN 通过自动识别图像的特征来进行特征的提取，并基于分类标签进行分类。本项目将借助 Keras 构建 CNN 网络，它是TensorFlow 的上层封装，提供了更多高级的 API 并且能够加快开发速度

2.2.2 迁移学习(Transfer learning)

迁移学习通俗地讲就是应用已有的知识来学习新的知识，在深度学习领域体现在把预训练的模型之参数搬迁至新模型中，以加快新模型的训练。因此，应用迁移学习时模型便不用从零开始学习（starting from scratch）。迁移学习要求预训练模型和新模型之间存在相似性。

本项目将应用迁移学习技术，使用预训练的 CNN 特征，具体就是使用在ImageNet 上大型卷积神经网络预训练的权重。因为 ImageNet 中的标签包含本项目中的分类标签——猫和狗，因此任务具有相关性，符合应用迁移学习的前提。

2.2.3 深度学习框架 Keras 和 GPU 计算

Keras 是一个高层神经网络 API，Keras 使用纯 Python 语言编写而成，并基于Tensorflow、Theano 以及 CNTK 后端。在本项目中使用的 Keras 版本为 2.1.5，且基于 Tensorflow-gpu 后端。Keras 让使用者能够简易和快速地进行原型设计，因为它具有高度模块化、极简和可扩充特性。

Keras 主要用于解决以下三类问题：

1）二分类问题（Binary Classification）；

2）类分类问题（Multi-class Clasification）；

3）标量回归（Scalar Regression）

GPU 加速计算是指同时利用图形处理器 (GPU) 和 CPU，以加快科学、分析、工程、消费和企业应用程序的运行速度。GPU 加速器于 2007 年由 NVIDIA 公司率先推出，现已在世界各地为政府实验室、高校、公司以及中小型企业的高能效数据中心提供支持。GPU 加速计算的原理是将应用程序计算密集部分的工作负载转移到 GPU 中运行，同时仍由 CPU 运行其余程序代码。

2.3模型介绍

在实践过程中，学习、尝试了多种预训练模型，下面对几种重要的模型进行介绍：

2.3.1VGG

VGG也称为VGGNet，是一种经典的卷积神经网络架构。VGG的开发是为了增加此类CNN的深度，以提高模型性能。它是具有多层的标准深度卷积神经网络架构。“深”是指由16和19个卷积层组成的VGG-16或VGG-19的层数。

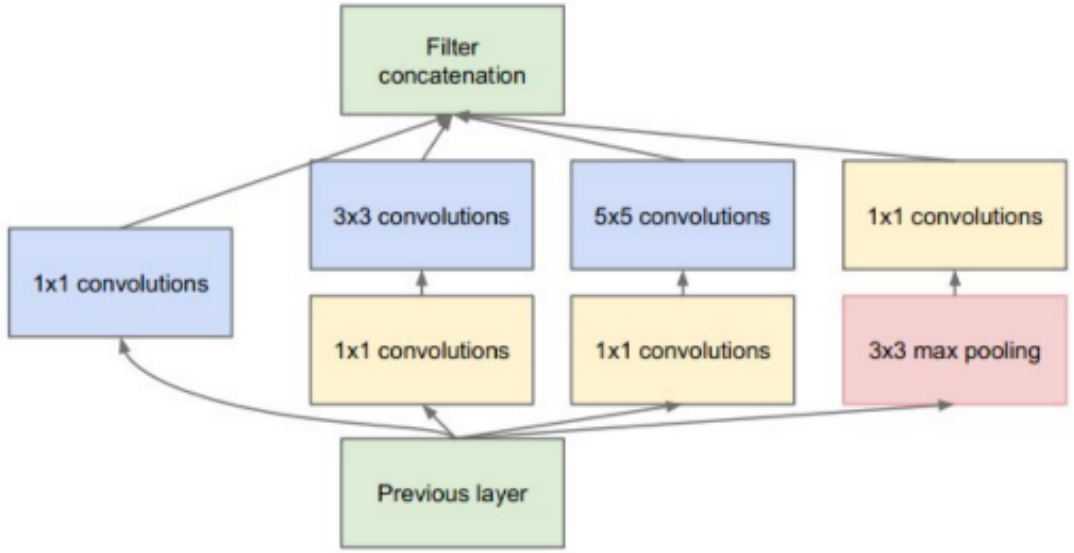
VGGNet使用的全部都是3x3的小卷积核和2x2的池化核，通过不断加深网络来提升性能。VGG可以通过重复使用简单的基础块来构建深度模型。

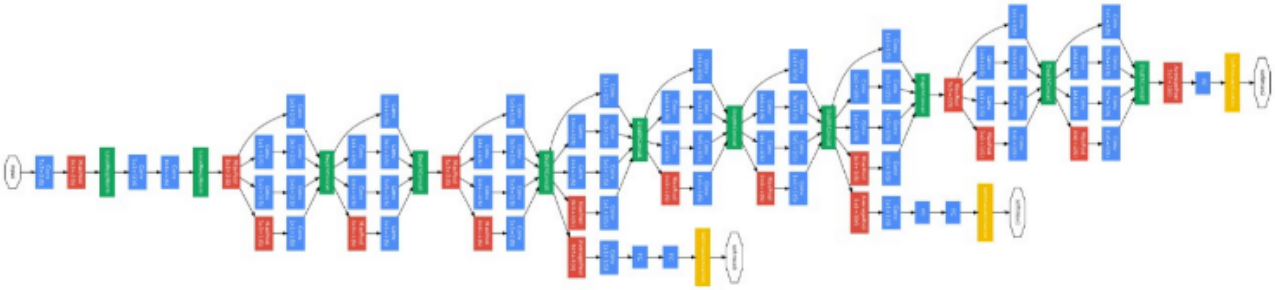
网络架构如图所示：

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

2.3.2Inception/GoogleNet

“Inception”微架构由 Szegedy 等人在 2014 年论文"Going Deeper withConvolutions"中首次提出。尽管 VGG 在 ImageNet 上表现很不错，但是其卷积层的通道数过大，需要的计算量是巨大的。例如一个 3×3 的卷积核，若输入输出的通道数都为 512，那么计算量将是 9×512×512。我们知道在卷积操作中，输出特征图和输出特征图的位置具有映射关系。但是，深度卷积网络中大部分的激活值是不必要的，即前面说的映射关系有大部分是冗余的。Inception 的搭建便是基于这样的理念，它体现了最高效的深度网络架构应该是激活值之间为稀疏链接的。虽然存在一些技术可以对网络进行剪至来获得稀疏权重，但是稀疏卷积核的乘法并没有得到优化，反而更慢。因此，GoogleNet 设计了一种 Inception 模块，其巧妙之处便是使用密集结构来近似一个稀疏的 CNN！另外，Inception 还使用 1×1 卷积核实现瓶颈层，而且最后还用全局池化层替换了全连接层，这两种做法都降低了计算量。





2.3.3ResNet模型

ResNet（Residual Neural Network）由微软研究院的Kaiming He等四名华人提出，通过使用ResNet Unit成功训练出了152层的神经网络，并在ILSVRC2015比赛中取得冠军，在top5上的错误率为3.57%，同时参数量比VGGNet低，效果非常明显。

模型的创新点在于提出残差学习的思想，在网络中增加了直连通道，将原始输入信息直接传到后面的层中，如下图所示：

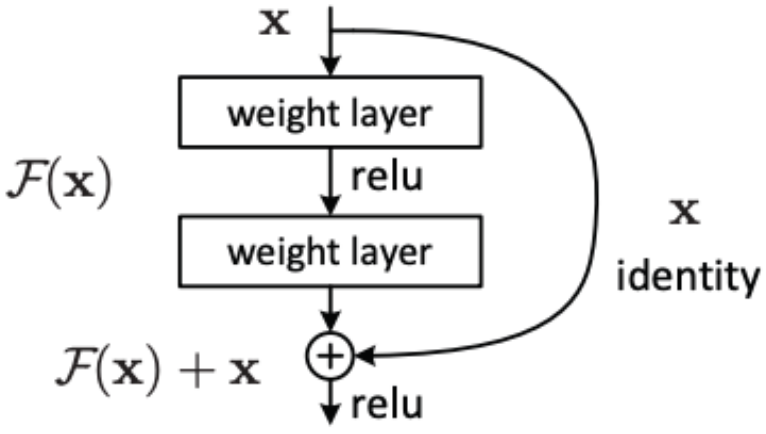
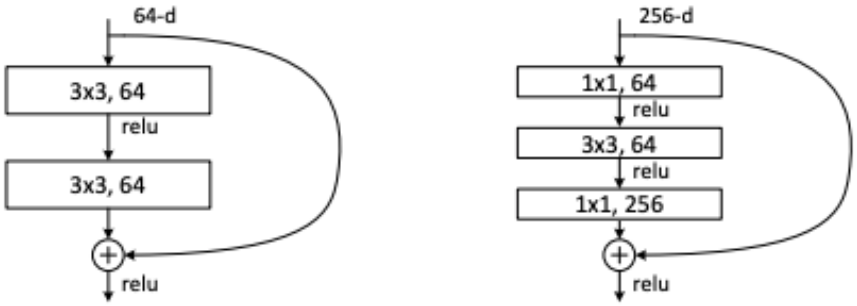


Figure 2. Residual learning: a building block.

传统的卷积网络或者全连接网络在信息传递的时候或多或少会存在信息丢失，损耗等问题，同时还有导致梯度消失或者梯度爆炸，导致很深的网络无法训练。ResNet在一定程度上解决了这个问题，通过直接将输入信息绕道传到输出，保护信息的完整性，整个网络只需要学习输入、输出差别的那一部分，简化学习目标和难度。ResNet最大的区别在于有很多的旁路将输入直接连接到后面的层，这种结构也被称为shortcut或者skip connections。

在ResNet网络结构中会用到两种残差模块，一种是以两个3*3的卷积网络串接在一起作为一个残差模块，另外一种是1*1、3*3、1*1的3个卷积网络串接在一起作为一个残差模块。如下图所示：



ResNet有不同的网络层数，比较常用的是18-layer，34-layer，50-layer，101-layer，152-layer。他们都是由上述的残差模块堆叠在一起实现的。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

本次使用ResNet18,34,50实现图像分类，模型使用pytorch集成的模型

从这些流行的 CNN 模型的结构设计可以看出，CNN 的设计趋向于模块化和高效率，特别是在 Inception 和 ResNet 中

三、实验实施

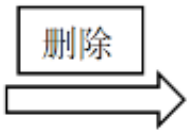
3.1实验方法

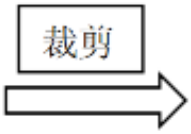
3.1.1数据预处理

尺寸异常值不会对模型的学习带来负面影响，但是数据中存在的标注异常值则不一样，**标注异常值**将影响模型的学习，必须最大限度地排除。处理基本思路是：

- **将与主题完全无关的图片删除；**
- **对于归类错误的图片，修改类别标注；**
- **将背景复杂的图片进行裁剪；**

这一步将完全通过人工方式手动处理。部分图片的处理方式如图所示：





另外，由于在使用预训练的模型时容易发生过拟合，因此在数据预处理阶段使用特征工程技术是一个十分可行的手段。特征工程（Feature Engineering）指的是在将数据输入模型之前，应用我们具备的机器学习算法知识和对数据的掌握，给数据加上一些硬编码。在该项目中，为了防止过拟合，我采用了数据增强技术。数据增强在针对图像分类问题中具体指，在训练的数据量相对较少时，通过平移、翻转、镜像和添加噪点等方法从已有的数据中创造出一批“新”的数据。

这样，模型在训练的时候，在每一个 epoch 中都不会面对完全一样的一批数据。虽然这些新增加的数据不能媲美原始训练数据（和原始训练数据存在关系），但对防止过拟合确实有帮助。

3.1.2特征提取

第一步，加载去掉顶层分类器的模型（留下卷积层）之结构和权重，并将训练数据输入入模型，提取特征向量；

第二步，为卷积层添加全连接层和 drop out 层，输入上一步提取的特征向量进行分类，观察分类结果并评价。

第三步，为卷积层添加一个包含卷积核的顶层分类器，将加载的卷积基础层冻结（不冻结顶层分类器的卷积核），重新编译模型。

输入图片数据进行训练，得到一个训练过的顶层分类器。之后，尝试使用参数微调方法。在使用特征提取手段时，我们已经得到了一个包含卷积层的顶层分类器。现在，可以将冻结的卷积层解开部分层，与顶层分类器一起训练。参数微调就是微调加载的模型的部分卷积层以及新加入的顶层分类器的权重。

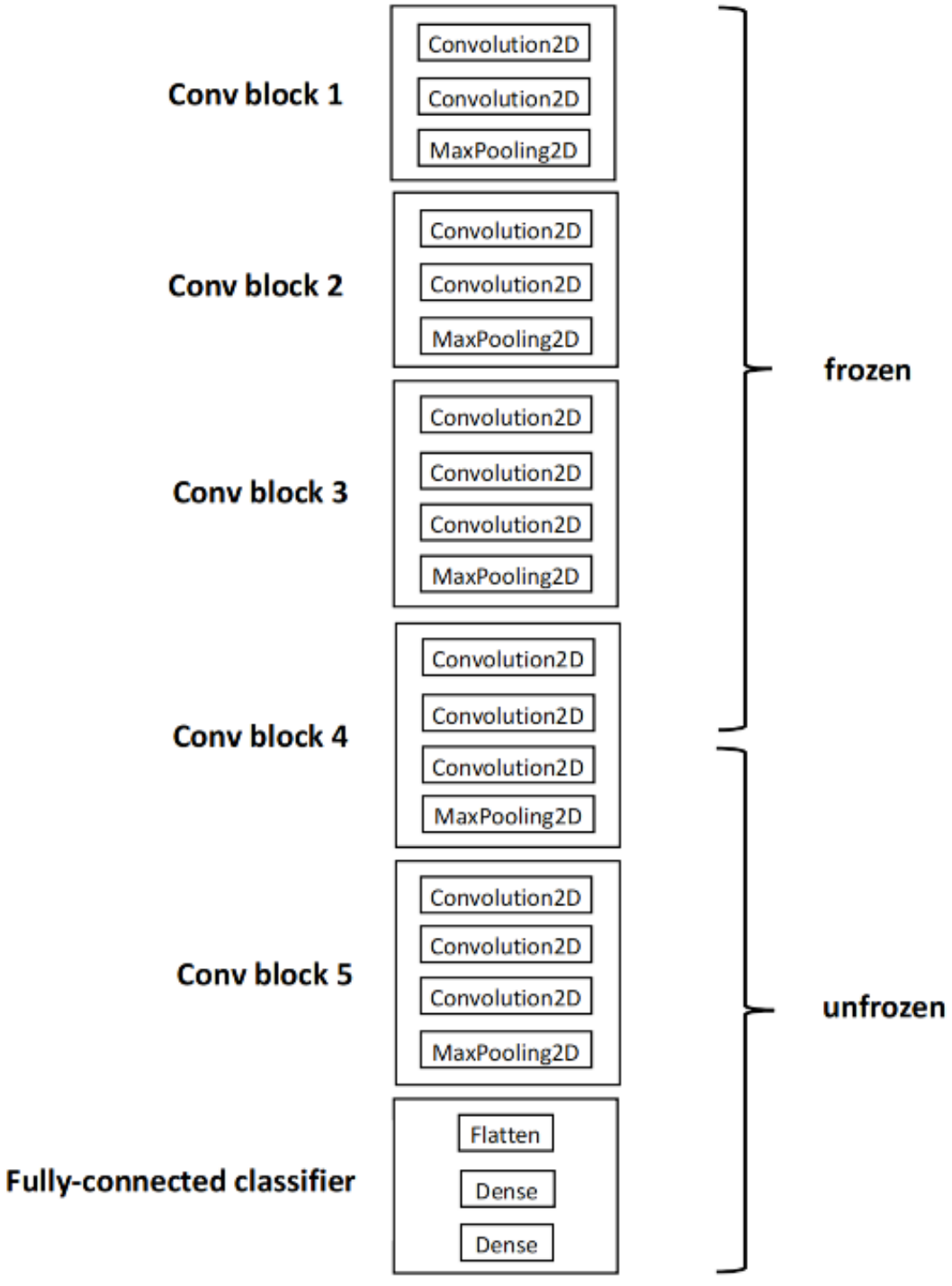
3.1.3参数微调

为了顺利地进行参数微调，模型的各个层都需要以预先训练过的权重为初始值。所以，不能将随机初始化权重的全连接层放在预训练过的卷积层之上，否则会破坏卷积层预训练获得的权重。体现在前面谈到的流程中，就是先使用特征提取方法训练顶层分类器，再基于这个顶层分类器进行参数微调。

另外，参数微调时不会选择训练整个网络的权重，而只微调位于模型中较深的部分卷积层，这在一定程度上可以防止过拟合。在前面已经提到，因为由底层卷积模块学习到的特征更具一般性，而不是抽象性。

对于 VGG16 模型，我首先尝试选择将其卷积模块的末尾 3 个卷积层冻结然后再进行参数微调，但效果并不理想。于是我又将冻结层扩大至末尾 4 个卷积层，进行参数微调；对于 InceptionV3，选择将 249 层之前的层冻结；对于 Resnet50，选择将 168 层之前的层冻结；对于 Xception，选择将 126 层之前的层冻结。

下图所示表示了 VGG16 参数微调的设置方法，其他模型的设置原理与此类似。



3.1.4训练

应用参数微调技术时应该在比较低的学习率下训练，这里使用的学习率为0.00001，优化器为 RMSprop。较低的学习率可以使训练过程中保持较低的更新幅度，以防止破坏模型卷积层预训练的特征。

在一开始，我将 Batch size 设置为 50，但是在训练时发现模型不容易收敛，而且在测试集和验证集上的损失函数值波动较大，引入 Keras 的回调函数EarlyStopping（EarlyStopping 能够让模型在训练时根据设定的条件停止）时，需将 patience 参数设置为较大的值。故后面将 Batch size 调整为较大值。但 Batch size太大对GPU显存资源也是一个很大的负荷，最终将训练时的 Batch size调整为 100。

为了应对模型过拟合问题，我使用了数据增强技术来训练一个新的网络，所以所有 epoch 中是不会有两次相同的输入的。但是，这些输入仍然是相互关联的，因为它们来自有限的原始图像，数据增强并不能产生新的信息，而只能重新混合现有的信息。因此，这可能不足以完全摆脱过度拟合。为了进一步克服过度拟合，我还将在模型中全连接层分类器的前面添加一个dropout层这是一种正则化手段，不过跟 Regularization 不同的是，它是通过将训练的层中的部分神经元的输出置零来实现的。在这里，使用的 Dropout 参数为 0.5。

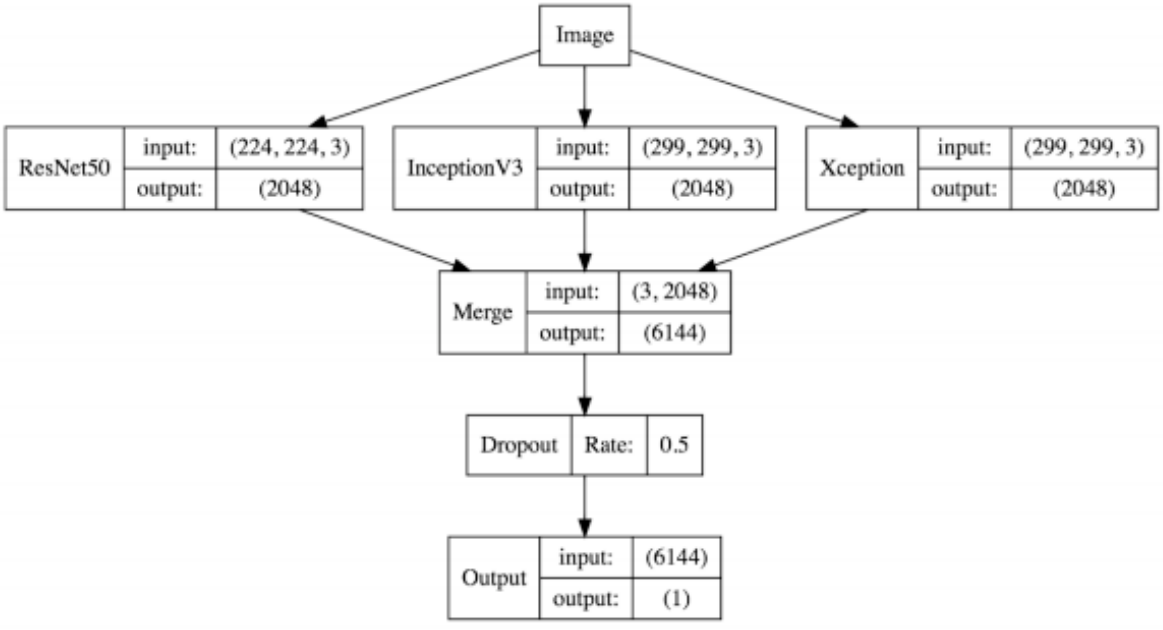
但在参数微调过程中，我发现 Xception 很快就停止训练了，因为使用了EarlyStopping 回调函数，而且 val_acc 和 val_loss 都呈现出模型的性能在下降的趋势，最后参数微调效果也不理想。于是我增大了数据增强的幅度，重新跑了一遍程序，这时 Xception 模型依然很快就停止训练，但是 val_acc 和 val_loss 却是往好的趋势变化。在这之后，我果断将 EarlyStopping 中的 stop 参数调整为较大的整数，从而增加 Xception 训练的 epoch 数量。

3.1.5 模型结合

使用预训练的模型进行参数微调确实比使用自己搭建的模型能够获得更小的损失函数值，但是训练过程还是十分缓慢的。在对每一个模型进行参数微调前，我都有都尝试直接使用特征向量来进行分类，而且个别模型的分类结果十分理想，包括 ResNet50、InceptionV3 和 Xception等。因此，尝试将提取的多个较优模型的特征进行组合，是一个非常可行的方案！

整个构建过程很简单。首先，我选择在前面的训练中表现好的三个模型ResNet50、InceptionV3 和 Xception 作为特征融合的对象模型。接着，先将数据输入各个模型中，提取特征向量并保存下来。然后，将对应同一个训练样本的来自这三个模型的三个特征向量在长度上进行堆叠，最终得到共 12500 个维度为 3 ×2048=6144 的特征向量。

然后，构建一个简单的神经网络，输入上一步特征融合得到的特征向量集合，进行训练。



然后尝试将GoogLeNet、ResNet和 ResNeX作为特征融合的对象模型，再进行上述训练。

3.2 核心代码介绍

3.2.1Pytorch预训练的模型：

数据增强：

为了提高模型的能力，使用pytorch自带的Transforms对图片进行处理变换。在训练时，可以对图片进行一定的剪裁，旋转。

```
# 对训练图片进行处理变换
my_transforms = transforms.Compose([
    transforms.Resize(75),
    transforms.RandomResizedCrop(64), #随机裁剪一个area然后再resize
    transforms.RandomHorizontalFlip(), #随机水平翻转
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# 对验证集的图片进行处理变换
valid_transforms = transforms.Compose([
    transforms.Resize((64,64)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

特征提取

采用迁移学习的思想，这里使用Pytorch预训练的模型“GoogLeNet”、“ResNet”和“ResNeXt”提取图像特征。

选择预训练模型的全局平均池化层的输出为新的特征，注意到对于每张图像，GoogLeNet提取到1024维特征；ResNet和ResNeXt提取到2048维特征；最后组合成5120维特征。

- 创建一个自定义的Net类，该类将预训练模型去掉最后的全连接层，以便用于特征抽取。
- 使用上述自定义的Net类对训练和测试数据进行特征抽取，并将特征和标签保存在相应的Tensor中。
- 最后，将特征数据和标签保存在HDF5文件中，每个模型对应一个HDF5文件，文件名为模型的名称

```
import torch
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import torchvision.models as models
import h5py

TRAIN_ROOT = 'data/train'
TEST_ROOT = 'data/test'
```

```
MEAN = [0.485, 0.456, 0.406] # ImageNet数据集的均值
STD = [0.229, 0.224, 0.225] # ImageNet数据集的标准差

def feature_extract(name):
    # 根据模型设置预处理方式
    if name == 'resnet50':
        p = 2048
        MODEL = models.resnet50(pretrained = True)
        print('resnet50 loads finish.')

    elif name == 'googlenet':
        p = 1024
        MODEL = models.googlenet(pretrained = True)
        print('googlenet loads finish.')

    elif name == 'resnext':
        p = 2048
        MODEL = models.resnext50_32x4d(pretrained = True)
        print('resnext50_32x4d loads finish.')

    preprocess = transforms.Compose([transforms.Resize(256),
transforms.CenterCrop(224),
                                     transforms.ToTensor(),
transforms.Normalize(mean=MEAN, std=STD)])

    # 加载数据
    train_data = ImageFolder(root = TRAIN_ROOT, transform = preprocess)
    test_data = ImageFolder(root = TEST_ROOT, transform = preprocess)
    # 对应的label
#     print(train_data.class_to_idx)

    # 冻结模型
    for param in MODEL.parameters():
        param.requires_grad = False

    # 去掉模型的全连接层
    class Net(torch.nn.Module):
        def __init__(self , model):
            super(Net, self).__init__()
            self.net_layer = torch.nn.Sequential(*list(model.children())[:-1])

        def forward(self, x):
            x = self.net_layer(x)
            return x.view(x.shape[0:2])

    model = Net(MODEL)
    model.eval()
```

```
    train_loader = DataLoader(train_data, batch_size = 100, shuffle = False, pin_memory
= True, num_workers = 2)
    test_loader = DataLoader(test_data, batch_size = 100, shuffle = False, pin_memory =
True, num_workers = 2)

    # 初始化要保存的数据
    train = torch.zeros(25000, p)
    test = torch.zeros(12500, p)
    label = torch.zeros(25000,)

    # 特征抽取
    with torch.no_grad():
        for i, (batch_x, batch_y) in enumerate(train_loader):
            output = model(batch_x)
            train[i*100:(i+1)*100] = output
            label[i*100:(i+1)*100] = batch_y
        for i, (batch_x, batch_y) in enumerate(test_loader):
            output = model(batch_x)
            test[i*100:(i+1)*100] = output
```

定义常量和设置随机种子：

这些常量定义了训练过程中需要用到的参数和超参数:

```
np.random.seed(2020)
torch.manual_seed(1)
torch.cuda.manual_seed(1)
batch_size = 100              # 每次训练的批量
hidden_units = 5120           # 全连接层隐藏单元数
num_labels = 2                # 输出类别数
dropout_rate = 0.5            # Dropout概率
learning_rate = 0.1           # 初始学习率
learning_rate_decay = 1       # 学习率衰减系数
lr_decat_step = 400           # 学习率衰减轮数
weight_decay = 5e-4           # 正则化系数
momentum = 0.9                # 动量系数
epochs = 8                    # 训练轮数
verbose = False               # 显示训练过程
train_from_scratch = True     # 是否从头开始训练
```

数据读取

```
# 配置 GPU/CPU 设备
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# 读取特征数据
```

```
def read_data():
    X_train = []
    X_pred = []

    for filename in ["googlenet.h5", "resnet50.h5", "resnext.h5"]:
        with h5py.File(filename, 'r') as h:
            X_train.append(np.array(h['train']))
            X_pred.append(np.array(h['test']))
            y_train = np.array(h['label'])

    X_train = np.concatenate(X_train, axis = 1)
    X_pred = np.concatenate(X_pred, axis = 1)
    assert X_train.shape == (25000, hidden_units)
    assert X_pred.shape == (12500, hidden_units)

    X_train, y_train = shuffle(X_train, y_train)
    return X_train, y_train, X_pred
```

配置模型

使用提取的特征作为输入进行二分类，直接用一个全连接层，输入5120维，输出2维(Softmax分类)

```
# 配置模型，是否继续上一次的训练
model = Net()
if train_from_scratch == False:
    model.load_state_dict(torch.load('model.pth'))
model.to(device)

# 设置优化器
optimizer = optim.SGD(model.parameters(), lr = learning_rate,
                      momentum = momentum, weight_decay=weight_decay)

# 设置学习率衰减
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size = lr_decat_step,
                                      gamma = learning_rate_decay)

# 设置损失函数
loss_func = nn.CrossEntropyLoss()
```

定义训练过程

用于定义模型的训练过程，包括前向传播、计算损失、反向传播和更新参数等操作;

```
# 定义训练过程
def train(train_loader, epoch):
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        output = model(data)
```

```
        scheduler.step()
        loss = loss_func(output, target)

        if batch_idx % 50 == 0 & verbose:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                    epoch, batch_idx * len(data), len(train_loader.dataset),
                    100. * batch_idx / len(train_loader), loss.item()))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

定义测试过程

定义模型的测试过程，计算模型在测试集上的损失和准确率;

```
def test(test_loader):
    test_loss = 0.
    correct = 0.
    model.eval()

    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += loss_func(output, target)
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()

        test_loss /= len(test_loader.dataset)
        print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
                test_loss.item(), correct, len(test_loader.dataset),
                100. * correct.type(torch.float) / len(test_loader.dataset)))
```

定义预测过程

进行模型的预测，并将结果保存到CSV文件中;

```
# 定义预测过程
def predict(X_pred):
    model.eval()

    with torch.no_grad():
        X_pred = X_pred.to(device)
        output = model(X_pred)

    output = output.argmax(dim = 1).cpu()
    y_pred = output.numpy().clip(0.01, 0.99)
```

```
    dataset = ImageFolder('data/test')
    df = pd.read_csv("sample_submission.csv")

    for i in range(len(y_pred)):
        fname = dataset.imgs[i][0]
        index = int(fname[fname.rfind('/')+1:fname.rfind('.')])
        df.label[index-1] = y_pred[i]

    df.to_csv('pred.csv', index = None)
```

参数搜索优化

定义了一个参数空间param_grid，其中包含了待搜索的参数及其取值范围。然后，构建了模型、优化器和学习率调度器。接下来，使用GridSearchCV进行参数搜索优化，设置参数为模型、参数空间、评估指标和交叉验证折数。最后，获取最佳参数组合和最佳准确率，并使用最佳参数进行训练和预测;

```
    # 设置参数搜索空间
    param_grid = {
        'net__module__lr': [0.01, 0.1, 1],
        'net__module__momentum': [0.9, 0.95, 0.99],
        'net__module__optimizer__step_size': [200, 400, 800],
        'net__module__optimizer__weight_decay': [1e-4, 5e-4, 1e-3]
    }

    # 构建Pipeline
    pipeline = Pipeline([
        ('pca', PCA(n_components=2)),
        ('net', NeuralNetClassifier(
            module=Net,
            max_epochs=epochs,
            optimizer=optim.SGD,
            callbacks=[EpochScoring('accuracy')],
            criterion=nn.CrossEntropyLoss(),
            device=device
        ))
    ])

    # 设置模型评估器
    gs = GridSearchCV(estimator=pipeline, param_grid=param_grid, scoring='accuracy',
cv=3, verbose=1)

    # 搜索最佳参数
    gs.fit(X_train, y_train)

    # 打印最佳参数和得分
    print('Best parameters: ', gs.best_params_)
    print('Best score: ', gs.best_score_)
```

3.2.2自己定义实现CNN模型进行预测：

- 第一层的卷积层conv1，卷积核(weights)的大小是 3*3, 输入的channel(管道数/深度)为3, 共有16个
- 第二层的卷积层cov2，卷积核(weights)的大小是 3*3, 输入的channel(管道数/深度)为16, 共有16个
- 第三层为全连接层local3,连接所有的特征, 将输出值给分类器 (将特征映射到样本标记空间), 该层映射出256个输出
- 第四层为全连接层local4,连接所有的特征, 将输出值给分类器 (将特征映射到样本标记空间), 该层映射出512个输出
- 第五层为输出层(回归层): softmax_linear,将前面的全连接层的输出，做一个线性回归，计算出每一类的得分，在这里是2类，所以这个层输出的是两个得分

```
def cnn_inference(images, batch_size, n_classes):


    with tf.variable_scope('conv1') as scope:

        weights = tf.get_variable('weights',
                                  shape=[3, 3, 3, 16],
                                  dtype=tf.float32,

 initializer=tf.truncated_normal_initializer(stddev=0.1, dtype=tf.float32))
        biases = tf.get_variable('biases',
                                 shape=[16],
                                 dtype=tf.float32,
                                 initializer=tf.constant_initializer(0.1))

        conv = tf.nn.conv2d(images, weights, strides=[1, 1, 1, 1], padding='SAME')
        pre_activation = tf.nn.bias_add(conv, biases)      # 加入偏差，biases向量与矩阵的每一行进行相加，shape不变
        conv1 = tf.nn.relu(pre_activation, name='conv1')   # 在conv1的命名空间里，用relu激活函数非线性化处理

    # 第一层的池化层pool1和规范化norm1(特征缩放)
    with tf.variable_scope('pooling1_lrn') as scope:
        # 对conv1池化得到feature map
        pool1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
                               padding='SAME', name='pooling1')
        # lrn()：局部响应归一化，一种防止过拟合的方法，增强了模型的泛化能力，
        norm1 = tf.nn.lrn(pool1, depth_radius=4, bias=1.0, alpha=0.001/9.0,
                          beta=0.75, name='norm1')


    # 第二层的卷积层cov2
    with tf.variable_scope('conv2') as scope:
        weights = tf.get_variable('weights',
                                  shape=[3, 3, 16, 16],  # 这里的第三位数字16需要等于上一层的tensor维度
                                  dtype=tf.float32,
```

```
initializer=tf.truncated_normal_initializer(stddev=0.1, dtype=tf.float32))
        biases = tf.get_variable('biases',
                                 shape=[16],
                                 dtype=tf.float32,
                                 initializer=tf.constant_initializer(0.1))
        conv = tf.nn.conv2d(norm1, weights, strides=[1, 1, 1, 1], padding='SAME')
        pre_activation = tf.nn.bias_add(conv, biases)
        conv2 = tf.nn.relu(pre_activation, name='conv2')

    # 第二层的池化层pool2和规范化norm2(特征缩放)
    with tf.variable_scope('pooling2_lrn') as scope:
        # 这里选择了先规范化再池化
        norm2 = tf.nn.lrn(conv2, depth_radius=4, bias=1.0, alpha=0.001/9.0,
                          beta=0.75, name='norm2')
        pool2 = tf.nn.max_pool(norm2, ksize=[1, 2, 2, 1], strides=[1, 1, 1, 1],
                               padding='SAME', name='pooling2')

    # 第三层为全连接层local3

    with tf.variable_scope('local3') as scope:
        # 将pool2张量铺平, 再把维度调整成shape(shape里的-1, 程序运行时会自动计算填充)
        reshape = tf.reshape(pool2, shape=[batch_size, -1])

        dim = reshape.get_shape()[1].value             # 获取reshape后的列数
        weights = tf.get_variable('weights',
                                  shape=[dim, 256],    # 连接256个神经元
                                  dtype=tf.float32,

initializer=tf.truncated_normal_initializer(stddev=0.005, dtype=tf.float32))
        biases = tf.get_variable('biases',
                                 shape=[256],
                                 dtype=tf.float32,
                                 initializer=tf.constant_initializer(0.1))
        # 矩阵相乘再加上biases, 用relu激活函数非线性化处理
        local3 = tf.nn.relu(tf.matmul(reshape, weights) + biases, name='local3')

    # 第四层为全连接层local4
    with tf.variable_scope('local4') as scope:
        weights = tf.get_variable('weights',
                                  shape=[256, 512],  # 再连接512个神经元
                                  dtype=tf.float32,

initializer=tf.truncated_normal_initializer(stddev=0.005, dtype=tf.float32))
        biases = tf.get_variable('biases',
                                 shape=[512],
                                 dtype=tf.float32,
                                 initializer=tf.constant_initializer(0.1))
        # 矩阵相乘再加上biases, 用relu激活函数非线性化处理
```

```
        local4 = tf.nn.relu(tf.matmul(local3, weights) + biases, name='local4')

    # 第五层为输出层(回归层): softmax_linear
    with tf.variable_scope('softmax_linear') as scope:
        weights = tf.get_variable('weights',
                                  shape=[512, n_classes],
                                  dtype=tf.float32,

 initializer=tf.truncated_normal_initializer(stddev=0.005, dtype=tf.float32))
        biases = tf.get_variable('biases',
                                 shape=[n_classes],
                                 dtype=tf.float32,
                                 initializer=tf.constant_initializer(0.1))

        # softmax_linear的行数=local4的行数，列数=weights的列数=bias的行数=需要分类的个数
        # 经过softmax函数用于分类过程中，它将多个神经元的输出，映射到（0,1）区间内，可以看成概率来理
解
        # 这里local4与weights矩阵相乘，再矩阵相加biases
        softmax_linear = tf.add(tf.matmul(local4, weights), biases,
name='softmax_linear')

    return softmax_linear
```

3.3环境介绍

运行环境：**MacBook Pro (15-inch, 2018)**

处理器： 2.6 GHz 六核Intel Core i7

语言环境：python3.7

3.4代码运行

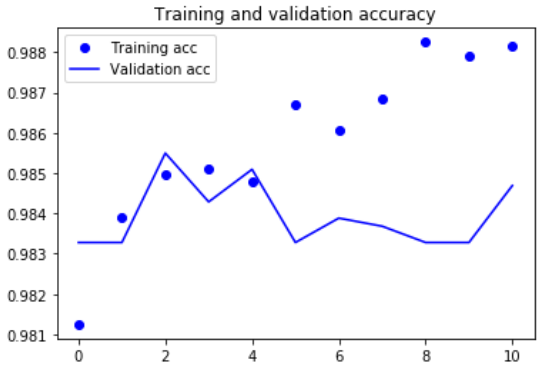
- 下载数据集，放入data文件夹
- 依次打开运行feature_extract.py文件，进行特征提取，也可以直接下载我提取好的，放入源文件

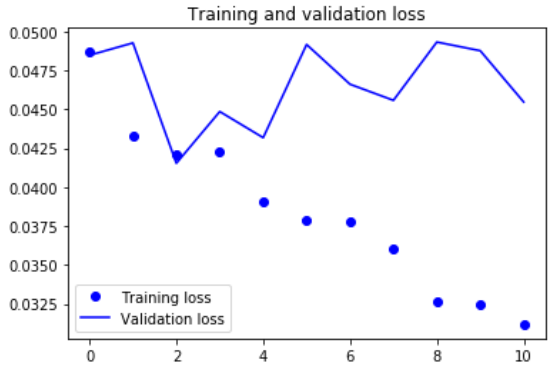
（链接: https://pan.baidu.com/s/1uC2wv_aBQiI3fk2EbPLLLg 提取码: teq9 ）

- 最后运行train.ipynb

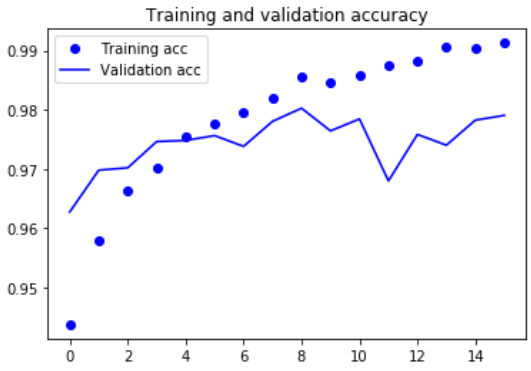
四、实验结果与分析

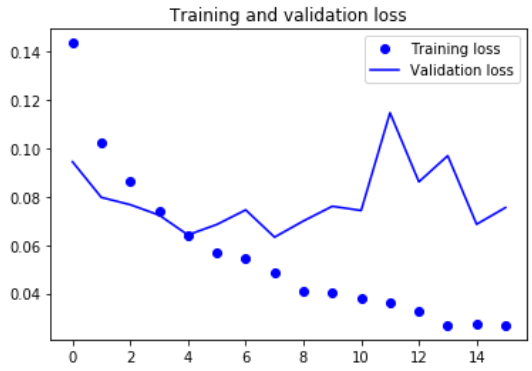
4.1模型准确率



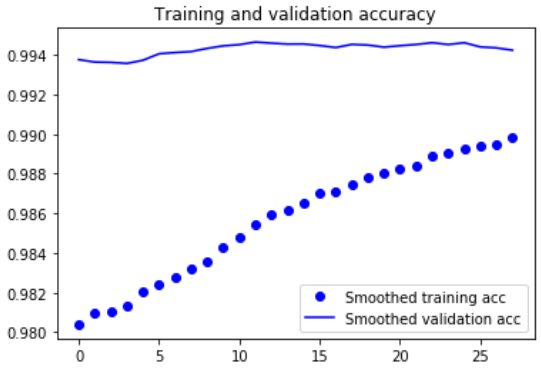


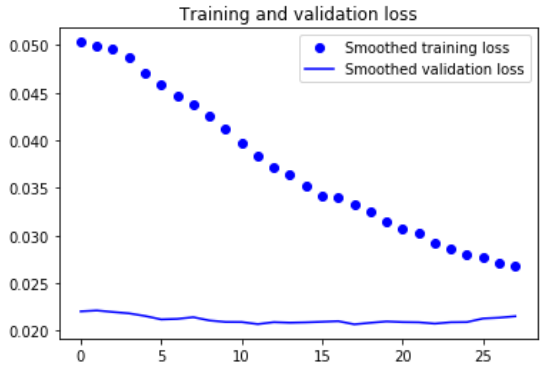
ResNet50



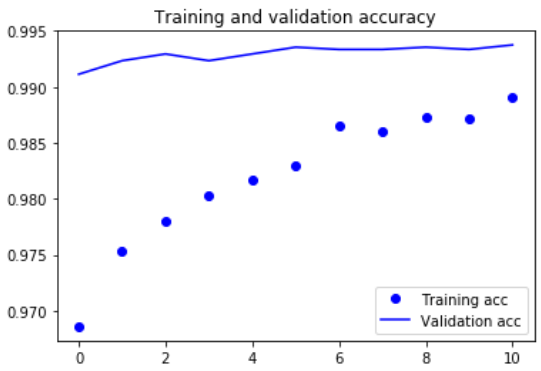


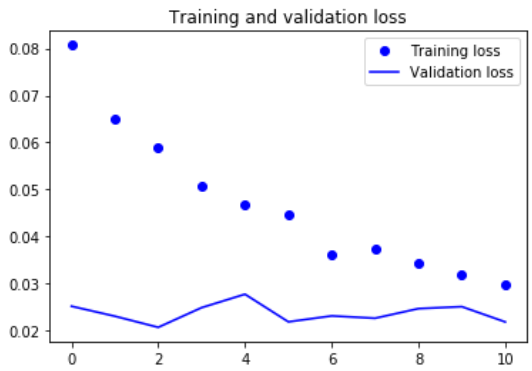
VGG16





Xception





InceptionV3

可以看到模型表现相对较优的是 Xception、Inception V3，它们在验证集上的分类准确率均高于 99%，损失值均低于 0.03。 而 Inception V3 在验证集上的分类准确率已经达到了 0.993%，logloss 值为0.025 左右，参数微调效果非常理想，和 Xception 十分接近！

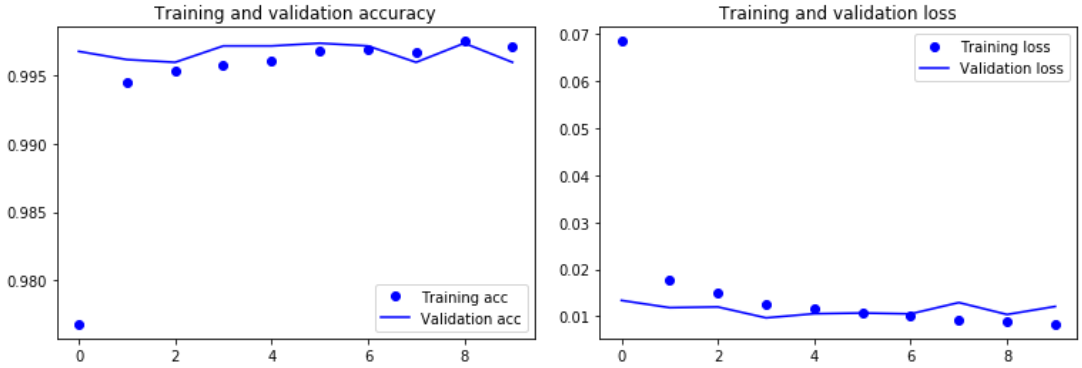
ResNet50 虽然也达到了不错的分类准确性，但结果却和 Inception V3 想比却处于劣势。初步猜测是模型训练时参数设置不够合理导致的，例如数据增强的设置，在往后的学习中会进行验证。

VGG16 的表现相对于前面三种模型则处于明显的劣势，毕竟 VGG16 相对较老，网络结构也较浅

4.2模型结合

将ResNet50、InceptionV3 和 Xception进行结合；

在验证集上的准确率很轻松地就达到了 **99.7%**， val_loss 也只有 0.012



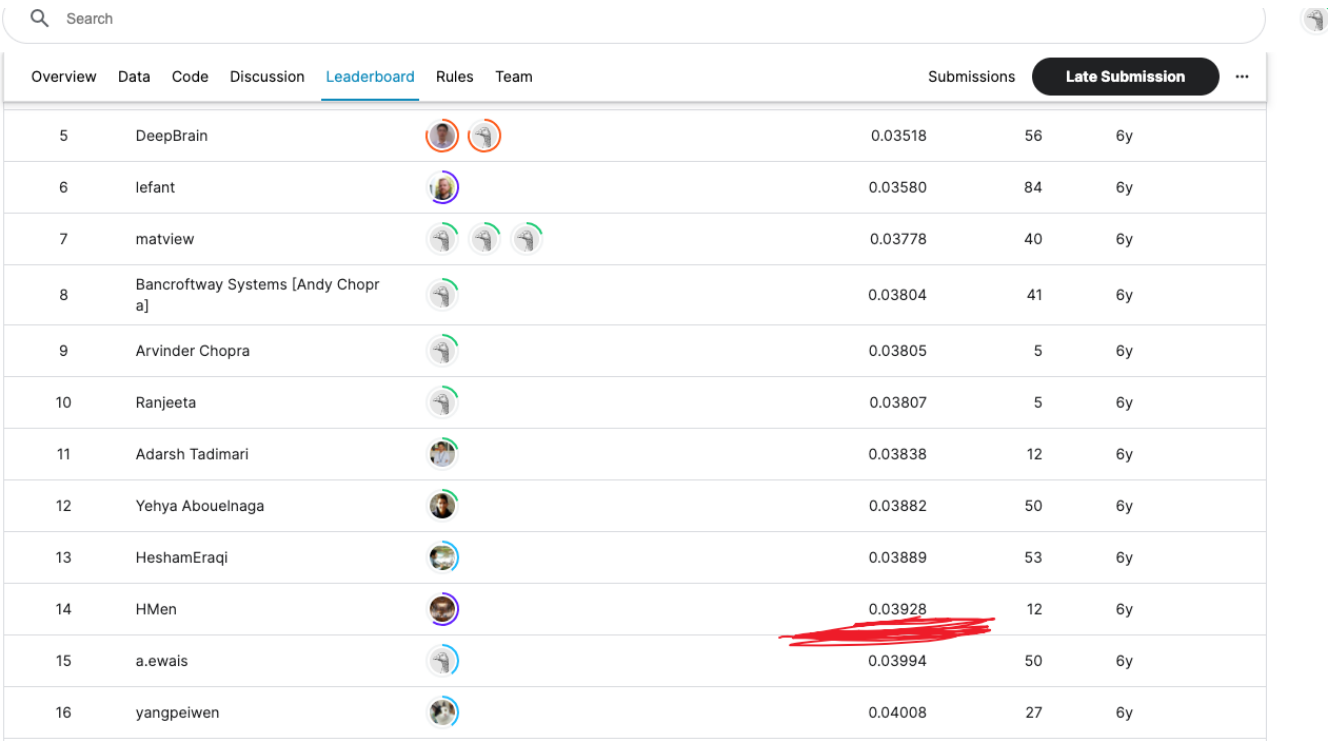
4.3预测结果优化

由于本项目的评价指标是对数损失函数 LogLoss，对于预测正样本，0.995 和 1 相差无几，但是对于预测负样本 0 和 0.005 却相差甚远。因此对模型预测输出结果进行截取可能会小幅地改善最终结果。所以在产生最后的预测结果数据时，都使用了 numpy.clip 函数将预测值限制在[0.005 和 0.995]这个区间。

将预测结果上传至Kaggle 网站中对应的猫狗分类比赛（https://www.kaggle.com/competitions/dogs-vs-cats-redux-kernels-edition/leaderboard），各模型获得的得分分别为：

Model	Score
VGG16	0.09509
Inception V3	0.04241
Resnet50	0.06505
Xception	0.04625
Feature Integration	0.0394

可见，在单模型中 Inception V3 和 Xception 的参数微调效果最好。Resnet50 次之，VGG16 与其他单模型相比差距较明显。而特征融合的得分最优，为 0.0394，在全球排名中可以排到 15/1314，比所有单模型的表现都要好。



4.4合理性分析

对单模型进行参数微调的合理性：对预训练模型进行参数微调属于迁移学习的一种方式。由于预训练模型先前在大数据集(这里加载的权重是在 Imagenet 训练得到的)上训练过，而且权重基本上是最优化的值，因此这些预训练模型本身就很可靠。如果从零开始设计和训练模型会花费很大的时间和精力，而且得到的结果不一定优于参数微调结果。

特征融合技术的合理性：特征融合技术的实现可以有多种方式。使用堆叠特征向量的方法，非常简单快捷，而且效果也比较理想

五、结论

5.1项目总结

在这次项目中，我对异常数据进行了处理，对进行了数据增强，学习、尝试了多种预训练模型、并进行参数微调，在训练过程中采用迁移学习的思想，对不同的模型提取的特征进行结合。

最终预测结果准确率在99.7%，提交到kaggle网站中对应的猫狗分类比赛，取得了15名的好成绩。

本次期末项目，包括代码编程、ppt制作、报告撰写，均是本人独立完成！

5.2个人总结

在图像分类领域，CNN 发挥着重大的作用。CNN 是一个非常灵活的模型，而且一直在进步，可以应用的新方法也在不断出现。同时，训练和调节 CNN 以获得更优的准确性也是一个充满挑战的过程，需要一定的技巧和充足的耐心。通过这个项目的实践，使得我对 CNN 的理解和应用水平得到了大步地提升，开始能够独立地写深度学习代码并解决实际问题。

需要做出的改进：

由于期末时间仓促，在进行参数微调时，各个模型冻结层和微调层的选择组合总数很有限。

如果要继续优化模型表现，可以使用更加复杂的特征融合技术。例如，使用参数微调后的预训练模型来导出特征向量，而不是直接使用从 Keras 中加载的预训练模型来导出。而且从特征融合训练曲线中可以看到，模型已经趋于过拟合，因此对于特征融合，还可以尝试在数据预处理阶段使用数据增强技术。

最后，可以手动地筛选分类错误的图片，针对具体模型，探索模型存在的缺陷。再根据这些缺陷，探究对模型进行进一步优化的方法。

六、参考论文

[1]He, Kaiming, et al. "Deep Residual Learning for Image Recognition." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016.

[2]Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Rethinking the Inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR) (pp. 2818-2826).

[3]Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR) (pp. 1251-1258).